Copyrighted material. Sun Microsystems, Inc. All rights reserved.

CHAPTER 3

Migration Strategies

This chapter defines the most important terms in migration and differentiates these terms. In addition, it presents migration strategies, the benefits and risks of each strategy, and the appropriateness of each strategy for various situations.

This chapter contains the following sections:

- "Understanding the Concepts" on page 21
- "Evaluating the Environment" on page 27
- "Examining Strategies" on page 30
- "Choosing a Strategy and Developing Tactics" on page 37

Understanding the Concepts

Within the context of this book, the term migration is defined as the transition of an environment's people, processes, or technologies from one implementation to another. While somewhat open ended, this definition allows us to discuss migration in a number of different contexts:

- The migration of common off-the-shelf (COTS) software from one platform to a larger or smaller similar platform
- The migration of data from one database to another, possibly similar data storage technology
- The migration of a custom-written application from one platform to a different platform and operating environment

These examples have different inputs, different execution strategies, and dissimilar functional outcomes, but they are all migrations in that the IT functionality has been moved from one platform to a different platform. It is this varied scope that affords the term "migration" its varied interpretations.

Consolidation

Migration is often confused with the act of *consolidation*. *Webster's College Dictionary* defines consolidation as, "the act of bringing together separate parts into a single or unified whole." As defined in the Sun BluePrints book *Consolidation in the Data Center*, by David Hornby and Ken Pepple, consolidation should be thought of as a way to reduce or minimize complexity. If you can reduce the number of devices you have to manage and the number of ways you manage them, your data center infrastructure will be simpler. This simplicity should contribute to efficiency, and consistency should contribute to lower management costs in the form of a reduced total cost of ownership (TCO).

While consolidations involve migrations as applications and business functionality are moved to a single machine, migrations do not necessarily involve consolidation. The value proposition realized by migration relates to improved quality of service (QoS) and reduced TCO realized as a characteristic of the new platform, environment, and overall IT infrastructure.

Adoption

Frequently, the release of a new edition of an operating environment might require that the platform's operating system (OS) be replaced with a more up-to-date version, a practice referred to as *adoption*. Depending on the changes introduced in the new version, upgrading to a new version of the current OS might be as difficult as migrating the application to a completely different OS, even though the hardware platform remains the same.

Usually, applications that must be moved to a newer version of the same environment have to be tested to determine whether they provide the same functionality in the new environment. This can be a time-consuming, as well as expensive, process. If test suites designed to verify critical application functionality are not available, they will need to be developed, at considerable time and expense.

Application programming interfaces (APIs) are the touch points between applications and the operating environment. In addition to defining APIs, the Solaris environment also supports the concept of the application binary interface (ABI), the description of these APIs used by the application executable at a binary level. This definition enables you to compare the API usage of an application executable created in one version of the OS to the binary description of interfaces supported by a different release of the OS. Consequently, compatibility between different versions of the OS can be guaranteed without any examination of the source code used to create the application.

In most cases, binaries created under an earlier version of the OS require no change. However, should any incompatibilities exist, specific APIs that don't conform to the ABI definition of the new OS can be identified. Tools exist to support the static

analysis of the application binary. Sun's `appcert` tool identifies differences so that they can be remediated prior to moving the application. This technology enables migration engineers to ascertain whether an application can be moved without problem to a newer version of the OS.

Moving applications from an older version of the Solaris OS to a newer version is referred to as *adoption* rather than migration. This distinction is made because the use of stable API standards, backward compatibility, and an ABI and tools enables you to verify and compare application interface usage, thereby guaranteeing that an application will run without problem in the new OS. In most cases, adoptions do not require recompilation, although using a later version of the compiler might provide performance benefits.

The E-Stack

Before we introduce migration and porting, consider the following figure, which illustrates what we call the Enterprise stack or E-stack.

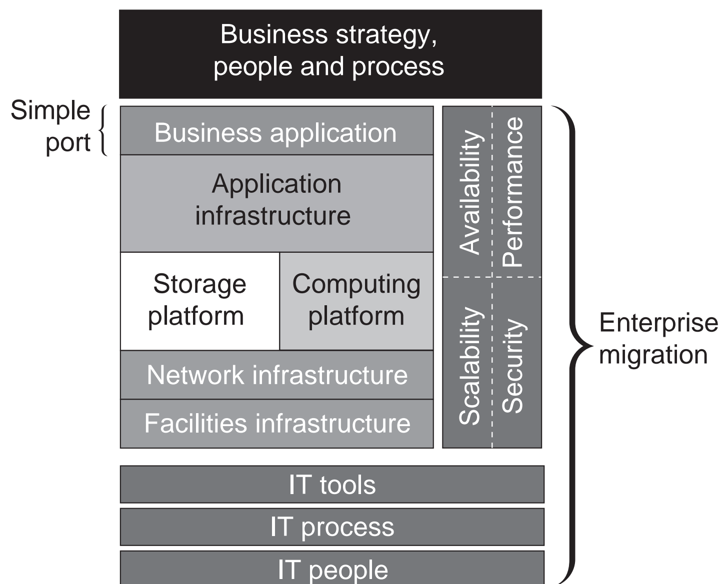

FIGURE 3-1 E-Stack Model for Enterprise Infrastructure

This construct is used as a model of the infrastructure of the enterprise. At the top of the stack, the business strategy, people, and process are defined. The high-level functions that occur here are usually controlled by an executive team. These functions provide logistical support to enable business functions, in addition to the

unique value-added functions that differentiate one enterprise from another. The outputs of this level of the stack must be implemented by the lower levels. Typically, applications are used to implement and execute business processes.

The next section of the E-stack represents the execution architecture, which is composed of the following items:

- **Applications that support business functionality.** Changes in business processes most likely require changes in the application layer. Consequently, a rapidly evolving business must be able to implement application change in a timely manner. Business segments that are not subject to frequent change are less likely to require an agile development or runtime environment to effect application change.
- **Application infrastructure that supports the applications.** Modern implementations of this layer include web servers, application servers, middleware, and database technology that support a typical n-tiered architecture. For older, legacy applications, the application infrastructure layer is composed almost entirely of the OS. Applications in this layer are written to interact with the APIs provided by the OS vendor and software provided by independent software vendors (ISVs).
- **Computing and storage platforms.** These hardware components enable the application infrastructure. This layer of the stack is usually composed of a heterogeneous mix of hardware provided by a number of different vendors. As we will see, industry consolidation within the hardware manufacturing segment might require changes at this level of the stack. Unless the vendor has gone to great lengths to support backward compatibility, changes to the computing and storage platforms will require changes to the layers above them in the stack.
- **Network infrastructure.** In today's environment, the ability to communicate over a network is critical. This infrastructure can be based on any type of networking technology, from low-speed dial-up to fiber-optic, high-speed backbones. Many legacy applications and their interfaces were designed and developed before the advent of networking technology. They depend on antiquated, proprietary interconnect technology, which might include a series of screens for data entry or possibly thick-client technology. These applications are not web enabled.
- **Facilities infrastructure.** Frequently overlooked, this layer provides critical support to the stack elements above.

The upper portion of the E-stack that defines the execution architecture supports a number of systemic properties that are key to any enterprise. Availability, scalability, measurability, and security are all desirable but can be implemented only to the extent that they are driven and supported by the execution architecture and its components.

The lower portion of the E-stack represents the management architecture. These tools, people, and processes implement the management infrastructure for the enterprise and combine to control, measure, and manage the execution architecture. Tools can be used to monitor capacity, utilization, and throughput, and to help

ensure that service levels can be met. Processes are in place to support change, service, deployment, and maintenance. These tools and processes are selected, developed, and administered by the IT staff. As change is effected within the E-stack, IT staff must be made aware of all changes. Training must take place to ensure that people understand the management process, as well as the execution architecture.

As you can see, all the elements of the stack support each other. If the facilities do not provide adequate power or air conditioning, the results will manifest themselves in the computing and storage platforms. If the computing and storage platforms do not support the application infrastructure, the application will not be able to function correctly and service level agreements will not be met. This would mean that the business function and requirements mandated and defined from the top layer of the stack could not be implemented.

The following table outlines the relationship between consolidation, migration, and adoption.

TABLE 3-1 Consolidation, Migration, and Adoption

Term	Definition
Consolidation	The act of reducing the complexity in a data center
Migration	The act of moving technology from one platform or OS to another
Adoption	The act of moving from an earlier version of the Solaris OS to a later version

Porting

As illustrated in the E-stack shown in FIGURE 3-1 on page 23, the term *porting* applies to applications rather than infrastructures. In particular, it is usually used in discussions about custom-written applications and refers to modifying or normalizing the code of an application so it can be recompiled and deployed on a new hardware platform that supports a different OS. Wherever possible, coding standards (ANSI, POSIX, and the like) should be adopted to minimize potential future changes that might have to be made.

Porting is inherently associated with modifying the code base of an application so that the functionality provided by the APIs of the existing OS and supporting software products is replicated in the new target environment. This is typically done by developing compatibility libraries that map the older APIs to the new environment. Vendors might provide these libraries to ease the burden of migrating applications to their environments, but in many cases, you will have to develop compatibility libraries yourself.

Porting the application requires minimal understanding of the logic or functionality of the application. It is a somewhat mechanical effort for making the application compatible with the new environment.

A porting strategy requires you to integrate the application with a new development environment, as well as with a new operating system. While source code, scripts, and data are moved, compilers, source code repositories, and software tools are replaced by new versions that are compatible with the target platform.

When porting an application, you must also migrate any supporting third-party software. If the software is no longer available, you will have to find similar software and integrate it into the application. Should the amount of integration become excessive, the migration might begin to look less like a port and more like a rearchitecture effort, as described later in the chapter. Ensure that you determine the availability of third-party software used by the application before choosing a migration strategy.

Migration

The term *enterprise migration* refers to the process of migrating all layers of the E-stack, not only the application that supports business functionality. This is a very involved exercise that will have a greater impact on the entire IT staff than other strategies do. For example, migrations include the following changes:

- Management policies present in the old environment must be migrated to the new environment.
- Tools used to monitor and manage the execution environment must be replicated.
- Supporting software, in the form of third-party products provided by an ISV, or locally written scripts to manage applications and data, must be integrated into the new environment.
- People must be trained to administer the new hardware platform and operating environment.
- Adding a large symmetric multiprocessor compute platform might justify the use of a multithreaded architecture for the application.
- Implementing a Storage Area Network (SAN) rather than attached storage might enable other applications to fully utilize storage resources that were previously unavailable.
- Adding networking capabilities might eliminate the need to transport and mount tapes.
- Web-enabling an application might reduce the need for proprietary terminal interfaces or thick-client technology.

- Changing hardware might require changes to the facilities to support more, or possibly less, power and cooling.
- Creating new tiers in the architecture might allow for the use of cheaper, more cost-effective hardware to support that portion of the application, which might, in turn, support greater availability and supportability.
- Using a modern programming language might enable the application to leverage more new third-party software, reducing the need for costly in-house development.

Evaluating the Environment

The migration solution you choose should be based on how an application fits into the overall IT environment. Consequently, you must evaluate the existing environment to determine both how the application meets business needs and how effective it is.

Adequacy of meeting business needs relates to the application's ability to support the business functionality of the enterprise. Adequacy of meeting business needs can be defined as follows:

- The time required to introduce new features
- The ease of use of the application
- The ability to support the functional requirements of the enterprise
- The ability to support the future growth of the enterprise

Whereas the adequacy of meeting business needs relates to the application's ability to meet the current and projected functional needs of the enterprise, IT effectiveness measures the application's use of technology. IT effectiveness can be defined as:

- Total cost of ownership (TCO)
- Technological stability
- Functional separation
- Service level issues
- Implementation technologies

The comparison of an application's IT effectiveness with the adequacy of meeting business needs is represented in the following figure.

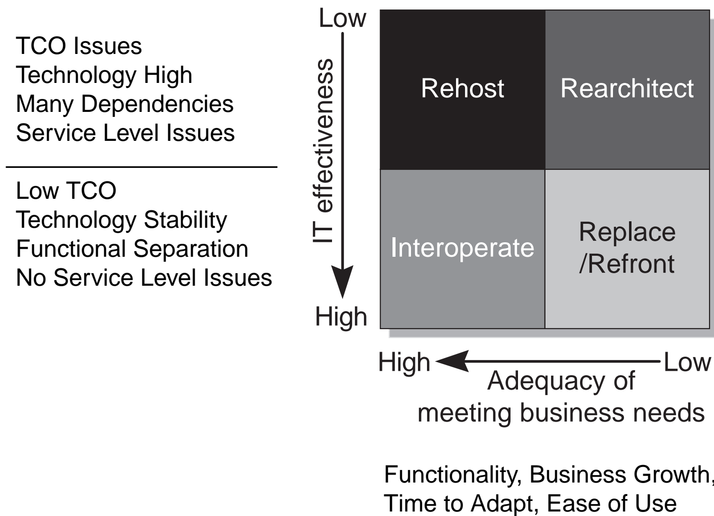

FIGURE 3-2 Effectiveness Versus Business Needs

The x axis evaluates how well the application currently fulfills its function in the business process. The y axis rates its IT effectiveness in terms of cost, technological stability, dependencies, and other factors. To determine a migration solution for each of the applications within the enterprise, you should begin by plotting the applications that have the greatest impact on the business process within this framework.

The evaluation of the application can be formal (for example, a complete TCO study), or it can be ad hoc. Typically, the Chief Financial Officer of the enterprise will have to agree with the evaluation before agreeing to a budget to support the migration. By systematically evaluating all applications with the same criteria, a comprehensive migration strategy can be developed that will include a number of different solutions, as illustrated in the example shown in the following figure.

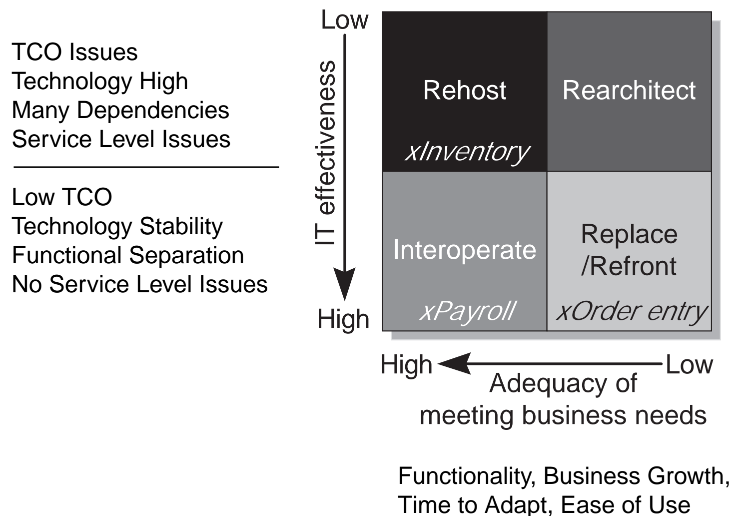

FIGURE 3-3 Example Evaluation of Applications

Applications with the following characteristics should be treated as follows:

- Applications that fall in the lower-left quadrant are meeting business process needs and are highly effective with respect to IT; they probably should be maintained as they are.
- Applications in the lower-right quadrant have high IT effectiveness, but are not meeting business process needs; they should be enhanced, not migrated.
- Applications that fall in the upper quadrants are the best candidates for migration. If they are meeting functional business process needs but are low in IT effectiveness (upper-left quadrant), it is probably time to move them to a new environment. If they exhibit low IT effectiveness and lack functionality (upper-right quadrant), it is probably time for the applications to be rearchitected.

FIGURE 3-3 illustrates how different applications (for example, payroll, inventory, and order entry applications) map to different migration solution spaces. For example:

- Order entry should be refronted because it relies on an antiquated user interface.
- The implementation and deployment of the payroll application is highly effective from an IT perspective and adequately meets business needs. It should remain where it is and be connected to new applications in the environment, as they are implemented, using connectors or adapters.
- The inventory application should be rehosted because the functionality it provides meets business needs, but there are issues with the hardware when it is implemented.

The terms “refront” and “rehost” are defined in detail in the following section.

Examining Strategies

In the following sections, we examine the various strategies available for a migration effort, including the following:

- Refronting
- Replacement
- Rehosting—technology porting
- Rearchitecting—reverse engineering
- Interoperation
- Retirement

When planning a migration project, consider how your environment could benefit from the strategies described in this section.

Refronting

Many legacy applications have excellent functionality but are not user friendly. Data entry for the application is accomplished by means of a series of screens that frequently contain cryptic names for fields and unintuitive menus, which result from limited screen space. These interfaces were based on CRT technology that was available 20 to 30 years ago.

Rather than rewriting an entire application, a programmer might be able to change just the data entry portion of the application. *Refronting*, or adding a more aesthetic interface to an existing application without changing the functionality, is an option. Users will have access to the same data but will be able to access it in a more efficient fashion without the use of expensive terminals, cabling, or peripheral interconnects.

When desired and appropriate, a browser-based solution can be developed. In the case of mainframe replacement, 3270 data entry screens can be replicated over a network. Web-enabling an application can provide significant cost reduction. Different approaches for refronting include screen scraping, HTML generation, source code porting, and other techniques.

Modern graphical user interface (GUI) technology can also be integrated into a legacy application to support a clearer representation of the required input. Conversely, for reasons of efficiency, the data entry screens might be replicated in the new technology “as is” to eliminate the need to train the data entry staff.

Some of the user acceptance issues identified by this example might reveal themselves when a rehost strategy is adopted and a COTS product upgrade involves a change to the input form's hosting technology (for example, when ASCII forms are replaced by a web browser).

The refront strategy requires an architectural model so that new components can invoke old, migrated components with minimum change to the migrated components. Such a migration project requires the application of architectural skills.

Replacement

The refronting strategy is really a variation of the much broader *replacement* strategy. With the replacement approach, the legacy application is decomposed into functional building blocks. Once an application is broken down in this manner, portions of a generic and often complex, custom-written legacy application can be replaced with a COTS application. Of course, the package must be able to run on the target OS.

When evaluating replacement strategies, consider packages that offer better functionality and robustness than do the existing, deployed application components. Make sure the vendor's solution is well tested and accepted in the marketplace, and verify that it is configurable, enhanceable, and well supported by the vendor. Product longevity and backward compatibility must also be taken into account.

One of the key drivers for the applicability of this strategy is the competitive dynamics of the software supply industry. Any custom (or bespoke) applications owned by users are always competing with the market, whether the competitive position is implicitly or explicitly evaluated. The marketplace is also driven by a sedimentation process. ISVs are seeking to maximize the value in business terms of their software products. Sedimentation refers to the moving of supporting functionality from the application implementation space to middleware (or utility software). From there, the functionality moves to the OS and often to hardware. For example, print spoolers and job schedulers are good examples of components that have been extracted from the application space and are now usually provided by utility software suppliers or by infrastructure vendors. Sometimes this moves even further, for example, in the case where web server load balancing functionality moved from the application layer to become an OS feature and is now implemented in networking hardware.

The sedimentation process is an opportunity for migration planners because it enables their state-of-the-art business functionality to become available to the enterprise. This occurs because the ISV developers can outsource the functionality development and maintenance to alternative providers and can concentrate on transactional logic. By migrating some functionality through the use of the replacement strategy, this trend can be copied, and the in-house code maintenance problem can be reduced, albeit through transfer to a third party. For example, cost

can be reduced or developer wages can be more focused on benefits, but cost is not eliminated. The replacement strategy enables in-house developers and maintainers of the utility code lines to be redeployed on more business-critical code lines and modules.

When considering replacement as a strategy, you might be attracted to the option to replace the application's code with a new third-party software product. If this approach is chosen, the migration project must do the following:

- Document the current business process and data model.
- Perform a gap analysis between the proposed application and the current state with respect to business process.
- Create a transformational data model, where appropriate.

These steps require traditional systems and business analysis skills.

A more powerful option might be to adopt a new application solution and change the organization's business logic when it no longer yields competitive advantage to match the package's optimum business process. This approach leaves migrators with the problem of identifying the still-used legacy data and migrating it to the new software solution. It also requires the development of a rollout plan that encompasses the enterprise's user community. Such a rollout is likely to be expensive, so the cost/benefit analysis of this approach needs to be solid and substantial. This analysis involves data modeling skills and, potentially, programs for transforming the data into the target data model and populating the new database. The latter approach allows the enterprise to transform applications built to deliver functional competitive advantage to software built to allow the user organization to compete through superior cost advantage. This goal mandates that the replacement product be competitively inexpensive to deploy and run.

Replacement can be a quick, low-risk solution, although the replacement of complete applications will have large implications in terms of business acceptance and rollout. However, replacing a homegrown solution with a COTS package can also take upward of two years. Effort is required to ensure that business processes and logic now conform to the capabilities of the COTS component, rather than the other way around. For some applications, the cost of acquiring custom logic for these software packages can be equivalent to maintaining and modifying a custom code base, depending on the function provided by that package, which is why it might be more appropriate to adopt the COTS vendors' assumed business process. Not all business processes, and hence not all applications, are designed to enable functional competitive advantage. For instance, a customer relationship management (CRM) package deployed to replace a specific business function will require more maintenance than configuring a replacement print spooler. If the proposed source modules for the migration are only a subset of the target package's functionality, it might make more sense to identify additional business processes to encapsulate within the CRM solution. For example, you might replace more code and increase the potential benefits case. This example shows the trade-offs available when replacement strategy-based migrations are being planned.

There are three clear options within the replacement strategy:

- Use a COTS package to replace or retire the source modules.
- Use a COTS/utility package to replace sedimented functionality.
- Use operating system functionality to replace sedimented application functionality.

The last option in the preceding list uses functionality that has been integrated into the existing OS. Examples of this include complicated memory management schemes that have been implemented because of older memory limitations, coarse-grained parallelism that is used instead of threading models, or shared memory that is used as an improved IPC shared memory.

The advantages of moving from a homegrown solution to a COTS-based solution include the following:

- Integration with other internal and third-party external applications
- The release of the budget associated with inflexible development resources
- An improved opportunity to tap skilled resources from established labor markets supporting both the business and IT communities

Replacement can act as a strategy on its own, and it can also be applied to components within an alternative strategy. Interestingly, as a strategy, it potentially yields the highest benefits and involves the highest degree of cost, yet when applied to components within an alternative strategy, it can be a quick and low-risk strategy.

Rehosting—Technology Porting

Rehosting involves moving complete applications from a legacy environment with no change in functionality. There are several ways this can be accomplished for custom-written applications:

- **Recompilation.** As previously mentioned, an application can be ported to the new environment. There are two approaches for doing this. The first approach is primarily associated with developing or acquiring a compatibility library that provides identical functionality to that of the APIs found on the original OS and that supports third-party products. For example, Sun provides compatibility libraries for some of the major competing operating systems such as HP/UX. An alternative approach is to use intelligent code transformation tools to alter the original source code to correctly call the new operating system's APIs. Both approaches have the benefit of capturing the changes required during the migration, although the second might limit backward compatibility.
- **Emulation.** This approach introduces an additional software layer to emulate the instruction set used in the source binaries. While introducing another software layer between the application and the hardware can affect performance, the new layer eliminates the need for recompilation. When adopting this strategy, it is

important to understand that the old environment has not really been left behind. The application will be developed and compiled using the old environment and will execute only in the new environment. By their nature, emulation solutions incur additional cost above that of the target platform environment. This results from the need to supplement the OS with the emulator, which is rarely free.

Although emulation is a useful approach, if source code is available, it is more common for an application to be recompiled to the native instruction set because native code runs faster. Emulation is most useful when migrating applications that are written in interpreted languages or when the original execution environment was tightly coupled within the OS. BASIC, PICK, or MUMPS are examples of environments that are suitable for emulation solutions.

Most emulators are for interpreted languages; therefore, the source code is available to the organization. However, source code engineering and reverse engineering rights might not have been granted in the right-to-use license. If you intend to use an emulation solution or reverse engineer a solution, ensure that you are licensed to do so in the environment where it will be used.

- **Technology porting.** Technology porting is a technique that supplements the target environment with the capability to execute code (usually interpreted) that runs natively on the original system. Many applications are developed and written in a superstructure software environment that is installed as a layered product on the source system. The most common types of these applications are created by relational database management system (RDBMS) vendors, many of whom support an array of hardware platforms and guarantee a common API across those platforms. The advantage of this approach is that one common API owner, the software ISV, owns the API on both the source and target systems. While the discovery stages of a migration project are still required, the APIs on the source and target systems remain the same.

  The leveraging of the ISV solution is often an opportunity to upgrade the ISV product version to obtain new functionality or to obtain superior support from the ISV. For instance, the transaction processing system known as CICS relies on a well-understood series of APIs. These APIs and their functionality have been ported or reimplemented on the new target Solaris OS. Applications using these APIs are compiled to run native instructions on the new system.

Rehosting offers the advantage of low development risk and enables familiar legacy applications to be quickly transferred to a more cost-effective platform that exhibits lower TCO and a faster return on investment (ROI). Extensive retraining of users is not needed because the architecture, interface, and functionality do not change. Rehosting is an excellent approach for companies desiring to decrease their maintenance and support costs.

Rehosting is, by definition, a quick fix. Rehosting does not change the application or the architecture. This means that new technology that is available in the target environment might not be properly utilized without some modification of the application. Rehosting is a preferred solution when the current business logic and

business process remain competitive in the enterprise's markets and are worth preserving. Rehosting offers the possibility of using cost savings accrued through switching development and runtime environments to fund full rearchitecture projects, when warranted.

Rearchitecting—Reverse Engineering

Rearchitecting is a tailored approach that enables the entire application architecture to migrate to the new OS, possibly using new programming paradigms and languages. Using this approach, applications are developed from scratch on a new platform, enabling organizations to significantly improve functionality and thereby take full advantage of the full potential of a target system.

Applications poor in IT effectiveness and functionality are the best candidates for rearchitecting. This approach is best used when time is not a major factor in the decision. Most rearchitecture projects require a skilled development staff that is well versed in the new technology to be implemented.

The downside to this approach is that it requires new or additional training for users, developers, and technical staff. In addition, rearchitecting requires the most time and is the most error prone of all the possible solutions. Sometimes, business rules can be well hidden in user interface or database management systems. For example, this was the case with DECforms, DEC FMS, and any RDBMS triggers. The ability to extract all the necessary business logic from the application's source can be severely inhibited by poor coding methodology and practice. An example is the hard-coding of business parameters.

Despite these problems, it remains that rearchitecture and reverse engineering are perceived to be the correct strategies and these problems become project risks. These risks can be mitigated by the application of appropriate business acceptance testing with internal and external users.

Rearchitecting does, however, open the opportunity to improve the business logic and processes and to change the developer productivity model.

A technique particularly appropriate to rearchitecture is reverse engineering. It is an axiom that the business logic encapsulated in the source code is the business logic implemented, and thus the source code is the most accurate place to discover the business logic. One of the key problems of software development is that most usability errors in software are introduced by poor business-process documentation and even poorer translation into software idioms. Some software environments have embedded dictionary or repository functionality. Where these exist, they may be supplemented with original author or third-party tools to enable the extraction of business logic and the recreation of that business logic in new environments. With these tools, the process can be reversed, the dictionary can be parsed, the user world view can be generated, and the implementation source code can be generated. A

classic example is the RDBMS world in which data definition language scripts for a database implementation can be generated from the database implementation itself. This is tool based, and tools might be proprietary to a single RDBMS or environment, or they might be open, running across multiple environments. Database schema generators are particularly useful for migrating from one RDBMS to another, such as from Microsoft's SQL Server to Oracle or Sybase.

Interoperation

In certain cases, it might be advantageous to leave an application where it is and surround it with new technology when it is required by an enterprise. *Interoperability* is a strategy that should be considered in the following cases:

- If business requirements are being met and IT effectiveness is high, it might be desirable to leave the application in its current environment, provided that environment is capable of interacting with current technology.
- Unfortunately, business drivers—for example, the existence of a leasing or outsourcing contract—might dictate that an application should stay where it is for some period of time. This is one of the risks of abandoning your IT environment to a third party. Over time, outsourced applications become orphans within the IT infrastructure. They are not fully integrated into the IT environment, most likely do not have a development staff, and run on outdated hardware that is no longer cost effective.

Many ISVs provide technology that enables legacy applications and storage technology to interoperate with newer technology. Intelligent adapters exist that support interactions between the mainframe and modern computing alternatives. It is also possible to compile an older language such as COBOL or PL/1 into Java™ bytecode, enabling it to seamlessly interact with a modern application server and other components of a Java™ 2 Enterprise Edition (J2EE™) environment.

When you choose this strategy, it is important to understand the vendor's commitment to the existing product line, as well as any future maintenance and product licensing costs. In addition, consider the availability of third-party software and current technological trends. When possible, open standards should be favored to allow a wide choice of competitive options.

Retirement

Changes in technology can obviate the need for specific functionality in an application or an overall solution. As middleware or third-party products mature, they might render the functionality implemented in the application obsolete. In this case, legacy utilities or legacy application functionalities can be retired because they are no longer required or are implemented elsewhere in the solution.

Choosing a Strategy and Developing Tactics

The ideal migration solution incorporates a number of the strategies listed in this chapter, where appropriate. Each of the strategies identified in the preceding section has a number of closely aligned supporting techniques. The selection of a strategy will define the obvious and most effective technique, but it might need to be supplemented with techniques more appropriate to other strategies.

The following table summarizes the strong alignments of particular migration techniques with the migration strategies.

TABLE 3-2 Strategy and Technique Alignment

Strategy	Complementary Technique
Refronting	Redeveloping, reverse engineering, source code porting
Replacement	Reverse engineering
Rehosting	Source code porting, technology porting
Rearchitecting	Reverse engineering, redeveloping
Interoperation	Technology porting, emulation
Retirement	Reverse engineering

As with the migration decision itself, the tactical approach used to solve a problem or to provide functionality must be evaluated in terms of its effectiveness, its impact on the ability to meet business needs, and its cost. Tactical decisions made to resolve technical issues might impact the overall project, either beneficially or adversely, in terms of systemic qualities, manageability, training, and cost.

As described in the preceding sections, there are a number of migration solutions for you to choose from. Each has its own benefits, as well as its own drawbacks. Selecting the correct solution should realize the associated value proposition.

Decision Factors

One factor in your decision should be based on your application architecture. Most modern application architectures are now based on n-tier models. This decomposition allows for different strategies to be applied to different tiers, when appropriate. This might mean that more than one strategy will drive your migration.

Conversely, older legacy applications might be monolithic or client-server in design and implementation, which offers the opportunity to rearchitect to an n-tier model. The following table outlines some common (not exclusive) approaches for each tier.

TABLE 3-3 n-Tier Migration Strategies

Tier	Purpose	Common Approaches
Presentation	Hosts the processing that adapts the display and interaction as appropriate for the accessing client device, be it a desktop computer, a cell phone, a PDA, or any other device.	Refronting, rehosting, interoperating, and replacing
Application or Business Logic	Hosts the logic that embodies the rules of the enterprise, irrespective of access device or resource implementation.	Rehosting, interoperating, and replacing
Integration	Allows for the connection of disparate applications and data sources.	Rehosting, interoperating, and replacing
Resource or Database	Consists of legacy systems, relational databases, data warehouses, or any other back-end or external processing system that accesses and organizes data.	Rehosting and replacing
Persistence	Holds the permanent data for the enterprise. In the past, this was considered part of the Resources tier, but with the growth of intelligent storage (SANs, NAS, and intelligent arrays), it has become a tier in itself.	Rehosting and replacing

Another factor to consider is the relationship between value and effort, as shown in the following figure. Typically, value is proportional to the amount of effort that is expended on a project. In the following paragraphs, we examine each of the proposed migration solutions as they relate to value versus effort.

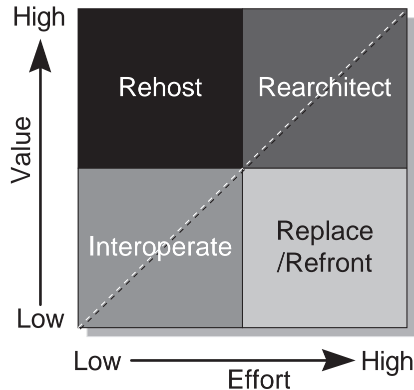

FIGURE 3-4 Relationship of Value to Effort

- **Interoperation.** This solution requires the least amount of effort but also provides the least amount of benefit. The existing architecture and infrastructure stay in place and simple connector technology is deployed to support the interaction with new applications or hardware that might be deployed. Because no new functionality is introduced, this effort requires minimal time and expense.
- **Rearchitecting.** This solution occupies the other end of the scale. Rearchitecting the application has great benefits: it supports tailored functionality; modular, tiered design; and a modern implementation language. However, the amount of effort (and associated cost) can be significant. As well as incurring significant expense, time, and effort, this solution can also introduce errors, so it requires a rigorous validation and verification effort.
- **Refronting or replacement.** This solution is targeted toward enhancing an application that has already been deemed to be somewhat unacceptable in meeting business needs. This solution is targeted to applications that are not meeting business needs but that do not have any IT-related problems relating to Service Level Agreements (SLAs), QoS, TCO, and the like. Enhancing the application by adding a presentation layer will add new functionality, but given that the application has already been found to be somewhat unacceptable, this enhancement adds minimal overall benefit compared with the amount of effort that it requires.
- **Rehosting.** As illustrated in FIGURE 3-4, rehosting is the solution that provides the most value for the least effort. Rehosting typically involves modifying the source code and build environment for an application so that it compiles and runs on the new target system. During this process, new features and functionality are not added. Many companies often want to add new features or functionality when they migrate, but these steps should take place after the application has been migrated.

Business logic remains the same when an application is rehosted. The only application changes usually relate to the APIs of the target OS. Over time, standards (SVR4, POSIX, and the like) have converged so that differences between versions of UNIX are minimal; therefore, migrations of applications between different versions of UNIX require minimal effort.

Rehosting applications from proprietary, non-UNIX environments that do not adhere to open standards can prove to be more challenging.

In certain cases, an application must be changed to not only adhere to the APIs of the OS but to interact with third-party product code as well. For example, consider the rehosting of a CICS application from MVS to MTP/MBM application running on the Solaris OS. The application's interaction with the MVS environment must be recoded in such a way that similar calls and functionality are used in the Solaris OS, but the CICS interaction requires minimal conversion because the CICS functionality and APIs have been redeveloped under the Solaris OS by the MTP and MBM product set.

Rehosting has the following characteristics:

- **Least expensive and requires the least effort.** Rehosting requires minimal changes to applications to enable them to run under the new environment. Therefore, the cost and effort involved in this strategy are minimal.
- **Quickest implementation.** Because little or no code is written and no new functionality is added, this solution can be completed in minimal time, compared with the other migration solutions.
- **Business logic remains the same.** Rehosting doesn't typically include the addition of new features or functionality. Consequently, the business logic remains the same, meaning that minimal or no staff training is required and few, if any, changes have to be made to the organizational structure.

Case Studies

In the last three chapters of this book, we examine three case studies, each of which uses a different migration strategy.

- **Case 1: Small business, Linux.** This example is based on a small software and services development company that is looking to move from Linux to the Solaris OS. Approximately 20 servers are being used: 10 for production, 5 for development and testing, and 5 for office support tasks. Their application is mostly Java-based, but they use MySQL for the database. Significant shell scripting has also been used for utilities in the product.

- **Case 2: Custom application, Tru64.** In this example, we examine the migration of a mythical inventory application implemented in the C programming language. The application is integrated with a Sybase database running under the Tru64 environment. The exercise involves porting the application so that it runs under the Solaris environment and replacing the Sybase database with an Oracle relational database.
- **Case 3: General ledger, HP-UX.** In this example, an insurance company planned to move its accounting, risks, and claims software from HP/UX to the Solaris OS to achieve superior scalability against planned business and system growth. This exercise employs the rehost strategy, in which the technology porting approach is used to minimize risk and cost.